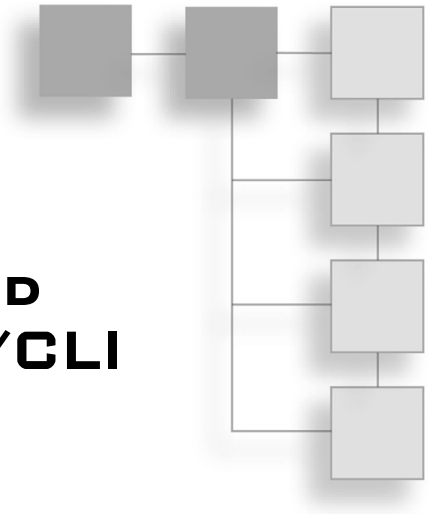
Bonus Chapter 2



Building a Managed Wrapper with C++/CLI

Act in haste and repent at leisure; code too soon and debug forever.

Raymond Kennington

The Microsoft .NET platform provides many tools and libraries to support rapid application development, code robustness, strict type safety, and much more. However, in order to take advantage of the features provided by this excellent platform, you must transition some, or all, of your existing code onto it. Some companies have the resources to port their existing solutions to .NET, but many companies (especially in game development) cannot spare the extra resources to migrate such a significant amount of work to the .NET platform. Typically, the largest amount of code for a game development studio is for the engine and its related subsystems.

A strongly sought after feature for custom tools and editors is direct integration with the engine instead of writing a watered-down version of the engine for functionality purposes. Exposing an engine to a toolset is quite difficult and problematic, especially when the engine is written in unmanaged C++ and the toolset is written in managed C#. It is not realistic to port the entire engine to the .NET framework for a number of reasons, such as performance and development time. Yet, tools developers are starting to see significant productivity and maintainability benefits from writing tools with .NET. The real trick is figuring out how to make both sides of the development team happy: tools developers and engine developers.

In this chapter, I first introduce C++/CLI (Managed C++), then describe what it is and what it is used for. I then introduce some rudimentary keywords and constructs used for working with managed code. Afterwards, a simple unmanaged code base is introduced and a wrapper built around it. The final part of this chapter shows how to wrap an unmanaged code base into a managed assembly with C++/CLI.

Introduction to Managed C++ (C++/CLI)

The .NET framework provides a set of extensions to the Visual C++ compiler and language to provide the ability to compile managed code and access the power and functionality of the .NET class framework. These extensions are known as C++/CLI (formerly known as Managed Extensions for C++), and include special keywords, attributes, preprocessor directives, and pragmas that facilitate the understanding of managed code. In addition to syntactical extensions, C++/CLI also offers a variety of additional compiler and linker settings to support managed compilation and the CLR. Even with the extensions, C++/CLI still follows the same rules for C++ syntax and keywords, except it follows .NET rules for types and architecture. C++/CLI can be thought of as "a language within a language."

Note

As of .NET 2.0, Managed Extensions for C++ has become known as C++/CLI and offers a redesign of the old syntax due to feedback from the development community. This chapter summarizes the new syntax and everything else inherited from the update to C++/CLI. You can still use the old syntax and functionality with the `/clr:oldSyntax` switch, but C++/CLI will be the only supported dialect moving forward, so it is important that you consider this migration.

C++/CLI provides mechanisms that allow managed and unmanaged code to coexist and interoperate in the same assembly or even in the same source file. No other language targeting the .NET runtime offers the interoperability features supported by C++/CLI.

Before continuing on into syntax, it is important to trace a mapping between native type names and their equivalent managed representations when compiled as managed code. Functions and methods are fairly automatic to wrap, but native types can present certain challenges. Value types are fairly standard, but the `System.String` object, for example, does not directly map to a `char` array.

Bonus Table 2.1 shows a listing of native types and their equivalent representations in managed code. If you are using the native type identifiers in managed C++, then you are actually aliasing the appropriate managed equivalent.

Bonus Table 2.1 Managed Equivalents of Native Types

Native Type	Managed Equivalent
bool	System.Boolean
signed char	System.SByte
unsigned char	System.Byte
wchar_t	System.Char
double	System.Double
long double	System.Double
float	System.Single
int	System.Int32
signed int	System.Int32
long	System.Int32
signed long	System.Int32
unsigned int	System.UInt32
unsigned long	System.UInt32
__int64	System.Int64
signed __int64	System.Int64
unsigned __int64	System.UInt64
short	System.Int16
signed short	System.Int16
unsigned short	System.UInt16
void	System.Void

This chapter only covers a small subset of the C++/CLI language. If you want to acquire comprehensive understanding of the language, I suggest you get the *C++/CLI Language Specification* file from Microsoft's Web site.

Overview of Extended Syntax

The syntax for the C++/CLI language is a couple hundred pages in length, so covering the entire language is unrealistic for this chapter. Even covering the changes from Managed Extensions for C++ to C++/CLI is significant in length, so this chapter will merely focus on the most important and commonly used changes with the new language. If you are new to C++/CLI but have prior experience with Managed C++, I recommend you download the C++/CLI Language Specification.

If you have never worked with Managed C++, I still recommend you download the specification and do not worry about learning the old syntax. Standard syntax related to unmanaged C++, carried over to C++/CLI, will not be covered in this section (such as type accessibility with the `public` and `private` keywords).

Reference Handles

Perhaps one of the most confusing elements of the old Managed C++ syntax was the sharing of the * punctuator for unmanaged pointers and managed references. The syntax for C++/CLI has cleaned up many aspects of the language, including the introduction of reference handles. Reference handles are managed references to objects that are located on the managed heap and are expressed with the ^ punctuator (pronounced "cap"). Handles are completely different from pointers, which just reference a particular location in memory. Pointers are unaffected by the garbage collector, but this also means that the garbage collector cannot optimize memory storage. Reference handles point to the object on the managed heap, so memory can move around and handles will update accordingly. Developers must explicitly delete memory when using pointers, whereas explicitly deleting is optional when using reference handles.

The following code shows a simple way to create a string, referenced by a handle.

```
String^ myString = "Hello World";
```

Just as `new` returns a pointer, `gcnew` returns a reference handle. Using `gcnew` offers an easy way to differentiate between managed and unmanaged instantiations. The following code shows `gcnew` returning a reference handle.

```
Object^ myObject = gcnew Object();
```

Note

Handles are type-safe, which means you cannot cast a handle to a `void^`.

In Managed Extensions for C++, reference types were prefaced with the `__gc` keyword. In the new C++/CLI language, the `__gc` keyword is replaced by either `ref class` or `ref struct`, depending on the type needed. The following code shows how to declare a managed class.

```
ref class MyClass
{
    // … Class implementation here
};
```

Similarly, value types were originally prefaced with the `__value` keyword, but now you use either `value class` or `value struct`, depending on the type needed.

Keyword: abstract

The abstract is context-sensitive and is used to declare that a member can only be defined in a derived type and that a class cannot be instantiated directly. This keyword is also valid when compiling native targets.

The following code will generate a compile error (C3622) when executed because `MyBaseClass` is marked as abstract.

```
ref class MyBaseClass abstract
{
public:
    virtual void SimpleFunction() {}
};

int _tmain()
{
    MyBaseClass^ baseClass = gcnew MyBaseClass;
}
```

Also, a similar compile error (C3634) will be generated for each of the two functions in the following class when instantiated directly. One of the methods is marked as abstract, while the other method is declared to be pure virtual.

```
ref class MyBaseClass abstract
{
public:
    virtual void SimpleMethod abstract () {}
    virtual void OtherMethod() = 0 {}
};
```

Note

Declaring an abstract function is the same as declaring a pure virtual function. Also, the enclosing class is also declared as abstract if a member function is declared to be abstract.

Keyword: delegate

Programmers who have worked with C++ should be familiar with function pointers. A similar mechanism known as a delegate exists in the .NET world. A delegate is

basically a reference type that can encapsulate one or more methods that conform to a specific function prototype. The following code shows how to define a delegate.

```
public delegate void SimpleDelegate(int number);
```

Next, we will define a class that has a couple of methods conforming to our new delegate signature.

```
ref class SimpleClass
{
public:
    void SimpleMethod(int number)
    {
        // ... Do something special
    }

    void AnotherMethod(int anotherNumber)
    {
        // ... Do something else
    }
}
```

At this point, we can simply attach methods to this delegate and fire a call using it. The following code shows how to attach a method to the delegate and execute it. A `null` check is used on the delegate instance to make sure that there is at least one method bound to it. This variable will be `null` when no methods are attached to it, so be sure to test for `null`.

```
int main()
{
    SimpleClass^ simpleClass = gcnew SimpleClass;

    SimpleDelegate^ simpleDelegate = gcnew SimpleDelegate(simpleClass,
                                                &SimpleClass::SimpleMethod);

    if (simpleDelegate)
        simpleDelegate(12345);
}
```

You could attach a second method to the delegate by using the following code.

```
simpleDelegate += gcnew SimpleDelegate(simpleClass, &SimpleClass::AnotherMethod);
```

Similarly, you can remove an attached method by using the following code.

```
simpleDelegate -= gcnew SimpleDelegate(simpleClass, &SimpleClass::AnotherMethod);
```

Keyword: event

The .NET platform is largely event-driven, and being able to provide a way to handle notifications when a significant event occurs is extremely powerful. The `event` keyword is used to declare an event method of a managed class. Events in their simplest form are associations between delegates (essentially function pointers in C++) and member functions (essentially event handlers) that can handle fired events and respond to them appropriately. Clients create event handlers by registering methods that map to the signature of the specified delegate. The great thing about delegates is that they can have multiple methods registered to themselves. This allows you to factor in an event-driven model for your applications that is more or less a version of the observer pattern.

In C++/CLI, the first step is to create a delegate, unless the delegate is predefined in the .NET framework in certain situations. The following code defines the delegate that will be used as a simple event example.

```
public delegate void SimpleDelegateEventHandler(int number);
```

After creating a delegate, the next step is to create an event within a class that conforms to the delegate.

```
ref class SimpleClass
{
public:
    event SimpleDelegateEventHandler^ OnSimple;
};
```

Now that the event is defined, some functionality is usually placed within the event class in order to fire the event with the necessary parameters. The following code shows a revised version of the class.

```
ref class SimpleClass
{
public:
    event SimpleDelegateEventHandler^ OnSimple;

    void FireSimpleEvent(int number)
    {
```

```
        // Check if methods are attached
        // to the event (prevent exception)
        if (OnSimple)
            OnSimple(number);
    }
};
```

The definition of an event is now complete, so we can move on to attaching and calling this new event from client classes. The first thing to do is create a class that defines a method conforming to the delegate signature.

```
ref class SimpleClientClass
{
public:
    void MySimpleEventHandler(int number)
    {
        // … Do something
    }
};
```

With the event handler defined, we can move on to attaching this event handler to the event. For that, we need an instance of the class defining the event, and we need an instance of the class defining the event handler. Event handlers can be attached to an event or detached from an event with the += and -= operators, respectively.

The following code shows how to do this.

```
SimpleClass^ myEventClass = gcnew SimpleClass();
SimpleClientClient^ myEventHandlerClass = gcnew SimpleClientClient();
myEventClass->OnSimple += gcnew ClickEventHandler(myEventHandlerClass,
                                    &SimpleClientClient::MySimpleEventHandler);
myEventClass->FireSimpleEvent(12345);
```

Keyword: interface

An interface is used to define how a class may be implemented, and C++/CLI offers the ability to declare a managed interface with the `interface` keyword. Classes inherit (or more correctly, implement) interfaces, but it is important to know that an interface is not a class. Classes do not override methods defined in an interface; they implement them instead. Implementation of a member does not require a name lookup (or v-table), unlike overridden members of a class. An interface can define functions, properties, and events, all of which must have public

accessibility and be implemented by classes implementing the interface. You do not specify accessibility on interface member definitions, because they are automatically public. Interfaces can also define static data, members, events, functions, and properties. These static members must be defined and implemented in the interface.

The following code shows how to define and implement an interface.

```
public interface class ISimpleInterface
{
    property int SimpleProperty;
    void SimpleMethod();

    static void DoSomethingUseful()
    {
        // … Do something here
    }
}

public ref class SimpleClass : ISimpleInterface
{
private:
   int simpleProperty;

public:
    property int SimpleProperty
    {
        virtual int get() { return simpleProperty; }
        virtual void set(int value) { simpleProperty = value; }
    }

    virtual void SimpleMethod()
    {
        // … Do something here
    }
}
```

Note

The C# language only supports single inheritance, but multiple inheritance can be achieved through the clever use of interfaces.

Keyword: property

On a technical level, the Common Language Runtime only recognizes methods and fields; the closest thing to the concept of a property is nested types. Even though properties are not directly recognized as a type, metadata can be used by certain programming languages to convey the concept of properties. Technically speaking, a property is a member of its containing type. However, properties do not have any allocated space because they are basically references to the get and set methods representing the property. The compiler for each language generates the property metadata when the appropriate syntax is encountered. With first class support for properties, the C# compiler generates the get and set property methods, as well as the property metadata. Managed C++ (`/clr:oldSyntax`) does not have first class support for properties, which leads to ugly syntax and subtle bugs. The new language introduced with C++/CLI has first class support for properties, which offers the same clean syntax that the C# language has for properties.

The following code shows how to define a property in C++/CLI.

```
private:
   String^ simpleString;

public:
    property String^ SimpleString
    {
        String^ get()
        {
            return simpleString;
        }

        void set(String^ value)
        {
            simpleString = value;
        }
    }
```

For the most part, typical implementations follow the same simple pattern for get and set property methods. For the properties that do not implement business rules like value checking, you can use shorthand syntax to declare a property with default get and set methods and a private member variable.

The following code shows how to declare a property with shorthand syntax.

```
property String^ SimpleString;
```

Note

Keep in mind that this is easily expanded on later without breaking any interfaces, simply by expanding the construct at a later point.

Keyword: sealed

Sometimes it is desirable to prevent a class from being a base class, or to prevent a method from being overridden in a derived class. This is accomplished using the `sealed` keyword.

The following code shows how to seal a class from being derived.

```
ref class SimpleClass sealed
{
    // ... Class implementation here
};
```

The following code shows how to seal a method from being overridden.

```
ref class SimpleClass : BaseClass
{
public:
    virtual void DoSomething() sealed
    {
        // … Do something
    }
};
```

Note

The example class inherits from a base class because having a sealed virtual method on a base class is redundant; hence the inheritance.

With a sealed method in a base class, you cannot override the method as shown in the following code.

```
public ref class BaseClass
{
public:
    virtual void SimpleMethod() sealed
    {
        // … Do something
    }
```

```
};

public ref class SimpleClass sealed : BaseClass
{
public:
    virtual void SimpleMethod() override
    {
        // … Do something
    }
};
```

However, you can use the `new` modifier to specify a new implementation for a sealed method under the same name. The following code shows how to do this.

```
public ref class BaseClass
{
public:
    virtual void SimpleMethod() sealed
    {
        // … Do something
    }
};

public ref class SimpleClass sealed : BaseClass
{
public:
    virtual void SimpleMethod() override
    {
        // … Do something
    }
};
```

Keyword: __identifier

Sometimes you may require the ability to declare a class and use it, even though its name is a reserved language keyword. The `__identifier` keyword offers the ability to use a C++ keyword as an identifier.

The following code defines a class called `template` that is then instantiated with the `__identifier` keyword.

```
public ref class template
{
```

```
    // … Class Implementation
}

int main()
{
    __identifier(template)^ templateClass = gcnew __identifier(template)();
}
```

Note

Although using the `__identifier` keyword for identifiers that are not actually keywords is permitted, doing so is strongly discouraged.

Keyword: pin_ptr

One of the biggest challenges when interoperating with managed and unmanaged code is passing memory back and forth. Managed code stores memory in a garbage-collected heap where physical addresses of memory are never guaranteed to remain static throughout the lifetime of a managed application. In fact, you can almost be certain that the addresses will change (though the references will update the new location accordingly) when the garbage collector compacts unused memory blocks and performs optimization. Because of this, developers interoperating with managed and unmanaged code need a way to force the garbage collector to leave certain memory addresses where they are. This functionality is exposed through the `pin_ptr` keyword that lets you declare a pinning pointer, which is an interior pointer that stops the garbage collector from moving an object onto the managed heap. Pinning pointers are necessary so that managed memory addresses passed to unmanaged code will not unexpectedly change during resolution of the unmanaged context. An object is no longer pinned when the pinning pointer goes out of scope and no other pinning pointers reference the object.

The following code describes a simple unmanaged function that takes in an integer array and assigns values to the elements.

```
#pragma unmanaged
void NativeCall(int* numberArray, int arraySize)
{
    for (int index = 0; index < arraySize; index++)
    {
        numberArray[index] = index;
    }
}
```

The following code shows how to create a pinning pointer around a managed integer array and pass it into the `NativeCall` function.

```
using namespace System;
#pragma managed
public ref class SimpleClass
{
private:
    array<int>^ simpleArray;
public:
    SimpleClass()
    {
        simpleArray = gcnew array<int>(256);
    }

    public void CallNativeFunction()
    {
        // Created a pinning pointer at the first array element
        pin_ptr<int> pinnedPointer = &simpleArray[0];

        // Create a native pointer of the pinning pointer
        int* nativePointer = pinnedPointer;

        // Execute native call
        NativeCall(nativePointer);
    }
};
```

Pinning pointers can be used on reference handles, value types, boxed type handles, members of managed types, and on elements in a managed array; you cannot point pinning pointers to reference types. You also cannot use pinning pointers as function parameters, members of a class, target types for casting, or for the return type of a function. Pinning pointers cannot be declared as static variables; they must be declared as local variables on the stack. Because pinning pointers are basically a superset of native pointers, anything that can be assigned to a native pointer can also be assigned to a `pin_ptr`. Pinning pointers are able to perform the same operations that native pointers can perform, including pointer arithmetic and comparison.

Note

Pinning a member in an object has the effect of pinning the entire object.

Keyword: safe_cast

An extremely useful keyword in C++/CLI is `safe_cast`, which is the successor to `__try_cast`. This keyword provides the ability to change the type of an expression and generate MSIL code that is verifiable and safe. The `safe_cast` keyword accepts a reference type handle, value type, value type handle, or tracking reference to a value or reference type to use as the type ID, and `safe_cast` operates on any expression that can evaluate to one of the supported type IDs. This keyword will convert the result from the evaluated expression to the specified type ID. An exception will be thrown (`InvalidCastException`) if a safe cast cannot be performed.

The following code shows how to use the `safe_cast` keyword.

```
interface class ISword {};
interface class IShield {};

ref class BroadSword : public ISword {};

int main()
{
    ISword^ sword = gcnew BroadSword;

    try
    {
        // The following line throws an exception. IShield is not
        // implemented by BroadSword.
        IShield^ castTest = safe_cast<IShield^>(sword);

        // The following line successfully casts.
        BroadSword^ broadSword = safe_cast<BroadSword^>(sword);
    }
    catch (InvalidCastException^)
    {
        // Handle cast exceptions
    }
}
```

Note

You can use a `static_cast` in most places that a `safe_cast` can be used, but you are not guaranteed to end up with verifiable MSIL; `safe_cast` guarantees this.

Keyword: typeid

When working with managed code, there are many cases when you need to get the type information for a class or instance. C++/CLI allows you to get the `System::Type` for a type at compile time, similarly to getting a type at runtime using `GetType`. This functionality is provided to the C++/CLI language with the `typeid` keyword, which is the successor to `__typeof` in the old Managed C++ syntax (`/clr:oldSyntax`).

The following code shows how to get the `System::Type` object for a particular type.

```
ref class SimpleClass
{
    // … Simple class implementation here
};

int main()
{
    SimpleClass^ simpleClass = gcnew SimpleClass;

    Type^ simpleClassType1 = simpleClass::typeid;
    Type^ simpleClassType2 = simpleClass->GetType();

    if (simpleClassType1 == simpleClassType2)
    {
        Console.WriteLine("Both types are identical");
    }
}
```

Common Language Runtime Compilation (/clr)

The most important compiler option for C++/CLI is the `/clr` switch that specifies how applications and components will use the CLR. Just using the `/clr` switch as it is will enable your C++ application to use the managed runtime by creating metadata for your application that can be consumed by other CLR applications. This switch also allows your application to consume the data and types present in the metadata of other CLR applications. This is the default option for C++/CLI projects (also known as Mixed C++), and it allows assemblies to contain both unmanaged and managed parts, allowing them to take advantage of the features of .NET but still contain unmanaged code. Mixed assemblies make it much easier to update applications to use .NET features without requiring a complete rewrite.

Although Mixed C++ is the default option, it is one of three distinct types of components and applications. You can also create Pure C++ and Safe C++.

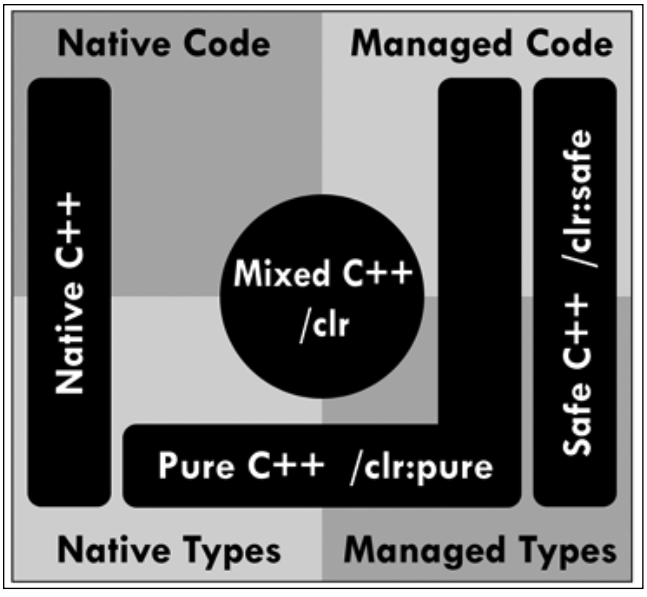
Pure C++ (`/clr:pure`) assemblies can contain both managed and unmanaged data types, but they can only contain managed code. P/Invoke can still be used to provide legacy interoperability, but you cannot use unmanaged C++ features with this option. One of the biggest advantages to using pure assemblies comes from performance gains. Pure assemblies can only contain MSIL, which means that no managed and unmanaged thunking occurs. Pure assemblies are also domain-aware, which makes interoperability between other .NET components easier and safer than mixed assemblies. Pure assemblies can also be loaded in memory or streamed, which is different from mixed assemblies, which are required to be located on the disk due to a dependency on the Windows loading mechanisms. Reflection is also severely dysfunctional in mixed assemblies, whereas pure assemblies have full reflection support. There are some disadvantages to using pure assemblies though. One reason, the most obvious, is that you cannot use pure assemblies to support interoperability with unmanaged code. This also means that pure assemblies cannot even implement COM interfaces or provide native callbacks. You also cannot use the ATL or MFC libraries, and the floating point options for exception handling and alignment are not adjustable.

Verifiable C++ (`/clr:safe`) assemblies are restricted to only containing code that can be guaranteed by the Common Language Runtime and code that does not violate any current security settings. You can enforce permissions, such as denying access to the file system, and these permissions will be enforced upon the verifiable assemblies by the CLR. It is important to consider building your libraries as verifiable code because future versions of the Windows operating system will start requiring that components and applications be verifiable and safe. Obviously, the benefit from using verifiable code is increased security, but you sacrifice CRT support and C++ interop features. You cannot use native code or data types in verifiable assemblies. You can, however, compile P/Invoke calls, but keep in mind that even if the code compiles you may still encounter security errors when you attempt to run the interop calls in an environment that disallows running them.

The three compilation types with the `/clr` switch are shown in Bonus Figure 2.1.

Note

You can compile your code with the old syntax from Managed Extensions for C++ by using the `/clr:oldSyntax` switch.



Bonus Figure 2.1 Available C++ compilation types.

Referencing Assemblies and Classes

Just like any other language targeting the .NET runtime, C++/CLI must add references to external managed components in order to gain access to their exposed functionality. In C++/CLI, assembly referencing is done with the `#using` directive. The syntax for this directive is `#using` *`file`* `[as_friend]`, where *`file`* can be a managed .dll, .exe, .netmodule, or .obj. An optional flag is `as_friend`, which is used to specify that all types within a file are accessible. Look into friend assemblies for more information on the type visibility of assemblies. Executable assemblies imported with the `#using` directive must be either compiled with `/clr:safe`, `/clr:pure`, or another verifiable language like C#. Trying to import metadata from an .exe assembly that has been compiled with `/clr` will result in an exception.

Assemblies references are located by either checking a path specified in the `#using` statement, the current working directory, the system directory of the .NET framework, directories added with the `/AI` compiler option, or directories added to the `LIBPATH` environment variable.

For custom assemblies, the syntax for the `#using` statement is

```
#using "YourAssembly.dll"
```

For framework assemblies, the syntax for the `#using` statement is

```
#using <System.Drawing.dll>
```

Note

When compiling with the `/clr` switch, the mscorlib.dll assembly is referenced automatically.

Mixing Managed and Unmanaged Code

C++/CLI supports the ability to mix managed and unmanaged code, which is a feature that is unique to C++/CLI; no other CLR language supports this feature. Managed and unmanaged code can be spread across many files within a single solution, but the really interesting feature is the ability to mix managed and unmanaged code within the same source file and within the same module. The compiler provides module-level control (`/clr` switch) for compiling either managed or unmanaged functions. Any unmanaged code will be compiled for the native platform, and execution of this code will be passed to native execution by the Common Language Runtime. By default, code in a C++/CLI project will be compiled to a managed target.

Mixing unmanaged and managed code in the same source file is accomplished through the use of the `#pragma managed` and `#pragma unmanaged` directives. Placing a `#pragma managed` directive before a function causes the function to be compiled as managed code, and you achieve an unmanaged effect using the `#pragma unmanaged` directive. These pragmas can precede a function, but cannot fall within the body of a function. Also, place the directives after `#include` statements, not before. The following code shows how to use the managed and unmanaged pragmas.

```
#using <mscorlib.dll>
using namespace System;
#include <stdio.h>

#pragma managed
void ManagedFunctionTest()
{
    printf("Hello Managed World!\n");
    ManagedFunction();
}

#pragma unmanaged
void UnmanagedFunctionTest()
```

```
{
    printf("Hello Unmanaged World!\n");
    ManagedFunctionTest();
}

#pragma managed
int main()
{
    UnmanagedFunctionTest();
    return 0;
}
```

Note

The compiler will ignore the managed and unmanaged pragmas when the `/clr` switch is not used.

It is also important to mention that when templates are involved, the pragma state at the time of definition is used to determine whether the code is managed or unmanaged.

Another little trick that can prove useful in certain situations is the ability to determine whether a source file is being compiled with the `/clr` switch or not. You can use the `__cplusplus_cli` preprocessor define to include or exclude code for different compile targets.

The following code shows how to include code when not compiling with the `/clr` switch.

```
#ifndef __cplusplus_cli
    // … Do something when compiling strictly for a native target
#endif
```

Example Unmanaged 3D Engine

In order to discuss interoperability and the creation of a wrapper, we need some sort of component or system to wrap. For this chapter, we will build a simple static library in unmanaged C++ that will create a Direct3D device and render a scene to it. In essence, we are building an extremely simple "rendering engine," which we will wrap into a managed library and consume in this chapter. There will not be much explanation for the base engine, because you should already have a basic understanding of the concepts and technology used. With that said, the full source code will be shown here so you can see the implementation details.

The following source code describes the header file of `SimpleEngine`.

```
#pragma once

#include <windows.h>
#include <tchar.h>
#include <d3d9.h>
#include <d3dx9.h>

//! Structure describing a vertex of the pyramid object
struct SimpleVertex
{
    //! The transformed(screen space) position for the vertex
    float X, Y, Z;

    //! The vertex color
    DWORD Color;
};

//! Describes the data format of the SimpleVertex structure
const DWORD SimpleVertexFVF = D3DFVF_XYZ | D3DFVF_DIFFUSE;

//! Simple class to provide a static lib
//! unmanaged 3D "engine" for the example
class __declspec(dllexport) CSimpleEngine
{
public:
    //! Constructor
    CSimpleEngine();

    //! Destructor
    ~CSimpleEngine();

public:
    //! Creates a Direct3D device and
    //! context using the specified window
    //! handle and dimensions
    HRESULT CreateContext(HWND window,
                          int width,
                          int height);

    //! Rebuilds the projection matrix of
    //! the device when the context resizes
```

```
        HRESULT ResizeContext(int width, int height);

        //! Renders the pyramid scene using the
        //! Direct3D render device
        HRESULT RenderContext();

        //! Used to release the device and
        //! Direct3D context resources
        HRESULT ReleaseContext();

    private:
        //! Initializes the device settings of the context
        HRESULT InitializeContext();

        //! Initializes the pyramid object resources
        HRESULT InitializeResources();

        //! Releases the pyramid object resources
        HRESULT ReleaseResources();

    public:
        //! Sets the color of the top corner pyramid vertex
        HRESULT SetColorTopCorner(DWORD color);

        //! Sets the color of the right front pyramid vertex
        HRESULT SetColorRightFront(DWORD color);

        //! Sets the color of the left front pyramid vertex
        HRESULT SetColorLeftFront(DWORD color);

        //! Sets the color of the back left pyramid vertex
        HRESULT SetColorBackLeft(DWORD color);

        //! Sets the color of the back right pyramid vertex
        HRESULT SetColorBackRight(DWORD color);

        //! Sets the color of the back buffer
        HRESULT SetColorBackBuffer(DWORD color);

        //! Gets the color of the top corner pyramid vertex
        DWORD GetColorTopCorner();
```

```
    //! Gets the color of the right front pyramid vertex
    DWORD GetColorRightFront();

    //! Gets the color of the left front pyramid vertex
    DWORD GetColorLeftFront();

    //! Gets the color of the back left pyramid vertex
    DWORD GetColorBackLeft();

    //! Gets the color of the back right pyramid vertex
    DWORD GetColorBackRight();

    //! Gets the color of the back buffer
    DWORD GetColorBackBuffer();

private:
    //! Direct3D object instance
    LPDIRECT3D9 direct3D;

    //! Direct3D device object instance
    LPDIRECT3DDEVICE9 device;

    //! Handle to the window that Direct3D will be bound
    HWND window;

    //! Vertex buffer for the pyramid object
    IDirect3DVertexBuffer9* vertexBuffer;

    //! Array describing the vertices making
    //! up the pyramid object
    SimpleVertex* vertices;

    //! Value describing the color of the top
    //! corner pyramid vertex
    DWORD colorTopCorner;

    //! Value describing the color of the right
    //! front pyramid vertex
    DWORD colorRightFront;

    //! Value describing the color of the left
    //! front pyramid vertex
    DWORD colorLeftFront;
```

```
    //! Value describing the color of the back
    //! left pyramid vertex
    DWORD colorBackLeft;

    //! Value describing the color of the back
    //! right pyramid vertex
    DWORD colorBackRight;

    //! Value describing the color of the back buffer
    DWORD colorBackBuffer;

    //! Flag specifying if the renderer
    //! is currently unavailable
    BOOL isLocked;
};
```

The following source code describes the source file of `SimpleEngine`.

```
#include "SimpleEngine.hpp"

#pragma comment(lib, "d3d9.lib")
#pragma comment(lib, "d3dx9.lib")

//! Constructor
CSimpleEngine::CSimpleEngine()
{
    colorTopCorner = 0xFFFF0000;
    colorRightFront = 0xFF0000FF;
    colorLeftFront = 0xFF00FF00;
    colorBackLeft = 0xFF0000FF;
    colorBackRight  = 0xFF00FF00;
    colorBackBuffer = 0xFF000000;

    vertexBuffer = NULL;
    vertices = NULL;

    isLocked = TRUE;
}

//! Destructor
CSimpleEngine::~CSimpleEngine()
{
    ReleaseContext();
}
```

```
//! Creates a Direct3D device and context
//! using the specified window handle and dimensions
HRESULT CSimpleEngine::CreateContext(HWND windowHandle,
                                     int width,
                                     int height)
{
    window = (HWND)windowHandle;

    RECT bounds;
    ::GetClientRect(window, &bounds);

    direct3D = Direct3DCreate9(D3D_SDK_VERSION);

    if (direct3D == NULL)
    {
        MessageBox(NULL,
                   _T("Can't find Direct3D SDK Version 9"),
                   _T("SimpleEngine Error"),
                   MB_OK | MB_ICONEXCLAMATION);
        return E_FAIL;
    }

    D3DPRESENT_PARAMETERS presentParams =
    {
        width,                          // Back Buffer Width
        height,                         // Back Buffer Height
        D3DFMT_R5G6B5,                  // Back Buffer Format (Color Depth)
        1,                              // Back Buffer Count (Double Buffer)
        D3DMULTISAMPLE_NONE,            // No Multi Sample Type
        0,                              // No Multi Sample Quality
        D3DSWAPEFFECT_DISCARD,          // Swap Effect (Fast)
        window,                         // The Window Handle
        TRUE,                           // Windowed or Fullscreen
        TRUE,                           // Enable Auto Depth Stencil
        D3DFMT_D16,                     // 16Bit Z-Buffer (Depth Buffer)
        0,                              // No Flags
        D3DPRESENT_RATE_DEFAULT,        // Default Refresh Rate
        D3DPRESENT_INTERVAL_DEFAULT     // Default Presentation Interval (V-Sync)
    };
```

```
    if (FAILED(direct3D->CheckDeviceFormat(D3DADAPTER_DEFAULT,
                                           D3DDEVTYPE_HAL,
                                           presentParams.BackBufferFormat,
                                           D3DUSAGE_DEPTHSTENCIL,
                                           D3DRTYPE_SURFACE,
                                           presentParams.AutoDepthStencilFormat)))
    {
        ::MessageBox(NULL,
                     _T("Cannot query specified surface format"),
                     _T("SimpleEngine Error"),
                     MB_OK | MB_ICONEXCLAMATION);
        return E_FAIL;
    }

    if (FAILED(direct3D->CreateDevice(D3DADAPTER_DEFAULT,
                                      D3DDEVTYPE_HAL,
                                      window,
                                      D3DCREATE_SOFTWARE_VERTEXPROCESSING,
                                      &presentParams,
                                      &device)))
    {
        ::MessageBox(NULL,
                     _T("Cannot create Direct3D device"),
                     _T("SimpleEngine Error"),
                     MB_OK | MB_ICONEXCLAMATION);
        return E_FAIL;
    }

    ResizeContext(width, height);

    InitializeResources();
    InitializeContext();

    return S_OK;
}

//! Rebuilds the projection matrix of the device when the context resizes
HRESULT CSimpleEngine::ResizeContext(int width, int height)
{
    if (height == 0)
        height = 1;
```

```
    D3DXMATRIXA16 projection;
    D3DXMatrixPerspectiveFovLH(&projection,
                               45.0f,
                               (float)width / (float)height,
                               0.1f,
                               100.0f);

    device->SetTransform(D3DTS_PROJECTION, &projection);
    D3DXMatrixIdentity(&projection);

    return S_OK;
}

//! Renders the pyramid scene using the Direct3D render device
HRESULT CSimpleEngine::RenderContext()
{
    if (isLocked != TRUE)
    {
        D3DXMATRIX view;
        D3DXMatrixLookAtLH(&view,
                           &D3DXVECTOR3(-5.0f, 0.5f, 16.0f),
                           &D3DXVECTOR3( 0.0f, 0.5f, 0.0f),
                           &D3DXVECTOR3( 0.0f, 1.0f, 0.0f));

        device->SetTransform(D3DTS_VIEW, &view);

        device->Clear(0,
                      NULL,
                      D3DCLEAR_TARGET | D3DCLEAR_ZBUFFER,
                      colorBackBuffer,
                      1.0f,
                      0);

        device->BeginScene();
        device->SetStreamSource(0,
                                vertexBuffer,
                                0,
                                sizeof(SimpleVertex));

        device->SetFVF(SimpleVertexFVF);
        device->DrawPrimitive(D3DPT_TRIANGLEFAN, 0, 4);
```

```
            device->EndScene();
            device->Present(NULL, NULL, NULL, NULL);
        }

        return S_OK;
    }

    //! Used to release the device and Direct3D context resources
    HRESULT CSimpleEngine::ReleaseContext()
    {
        ReleaseResources();

        if (device != NULL)
            device->Release();

        if (direct3D != NULL)
            direct3D->Release();

        device = NULL;
        direct3D = NULL;

        return S_OK;
    }

    //! Initializes the device settings of the context
    HRESULT CSimpleEngine::InitializeContext()
    {
        device->SetRenderState(D3DRS_ZENABLE,  TRUE );
        device->SetRenderState(D3DRS_CULLMODE, FALSE);
        device->SetRenderState(D3DRS_LIGHTING, FALSE);
        device->SetRenderState(D3DRS_SHADEMODE, D3DSHADE_PHONG);

        isLocked = FALSE;

        return S_OK;
    }

    //! Initializes the pyramid object resources
    HRESULT CSimpleEngine::InitializeResources()
    {
        ReleaseResources();
```

```
vertices = new SimpleVertex[6];
::ZeroMemory(vertices, sizeof(SimpleVertex) * 6);

vertices[0].X = -4.0f;
vertices[0].Y = 2.0f;
vertices[0].Z = 12.0f;
vertices[0].Color = colorTopCorner;

vertices[1].X = -3.0f;
vertices[1].Y = 0.0f;
vertices[1].Z = 11.0f;
vertices[1].Color = colorRightFront;

vertices[2].X = -5.0f;
vertices[2].Y = 0.0f;
vertices[2].Z = 11.0f;
vertices[2].Color = colorLeftFront;

vertices[3].X = -5.0f;
vertices[3].Y = 0.0f;
vertices[3].Z = 13.0f;
vertices[3].Color = colorBackLeft;

vertices[4].X = -3.0f;
vertices[4].Y = 0.0f;
vertices[4].Z = 13.0f;
vertices[4].Color = colorBackRight;

vertices[5].X = -3.0f;
vertices[5].Y = 0.0f;
vertices[5].Z = 11.0f;
vertices[5].Color = colorRightFront;

device->CreateVertexBuffer(6 * sizeof(SimpleVertex),
                           0,
                           SimpleVertexFVF,
                           D3DPOOL_DEFAULT,
                           &vertexBuffer,
                           NULL);

unsigned char* vertexData = NULL;
```

```
    vertexBuffer->Lock(0,
                       sizeof(SimpleVertex) * 6,
                       (void**)&vertexData,
                       0);
    {
        ::memcpy(vertexData, vertices, sizeof(SimpleVertex) * 6);
    }
    vertexBuffer->Unlock();

    isLocked = FALSE;

    return S_OK;
}

//! Releases the pyramid object resources
HRESULT CSimpleEngine::ReleaseResources()
{
    isLocked = TRUE;

    if (vertexBuffer != NULL)
    {
        vertexBuffer->Release();
        vertexBuffer = NULL;
    }

    delete [] vertices;
    vertices = NULL;

    return S_OK;
}

//! Sets the color of the top corner pyramid vertex
HRESULT CSimpleEngine::SetColorTopCorner(DWORD color)
{
    colorTopCorner = color;
    ReleaseResources();
    InitializeResources();
    return S_OK;
}

//! Sets the color of the right front pyramid vertex
HRESULT CSimpleEngine::SetColorRightFront(DWORD color)
{
```

```
    colorRightFront = color;
    ReleaseResources();
    InitializeResources();
    return S_OK;
}

//! Sets the color of the left front pyramid vertex
HRESULT CSimpleEngine::SetColorLeftFront(DWORD color)
{
    colorLeftFront = color;
    ReleaseResources();
    InitializeResources();
    return S_OK;
}

//! Sets the color of the back left pyramid vertex
HRESULT CSimpleEngine::SetColorBackLeft(DWORD color)
{
    colorBackLeft = color;
    ReleaseResources();
    InitializeResources();
    return S_OK;
}

//! Sets the color of the back right pyramid vertex
HRESULT CSimpleEngine::SetColorBackRight(DWORD color)
{
    colorBackRight = color;
    ReleaseResources();
    InitializeResources();
    return S_OK;
}

//! Sets the color of the back buffer
HRESULT CSimpleEngine::SetColorBackBuffer(DWORD color)
{
    colorBackBuffer = color;
    return S_OK;
}

//! Gets the color of the top corner pyramid vertex
DWORD CSimpleEngine::GetColorTopCorner()
{
```

```
    return colorTopCorner;
}

//! Gets the color of the right front pyramid vertex
DWORD CSimpleEngine::GetColorRightFront()
{
    return colorRightFront;
}

//! Gets the color of the left front pyramid vertex
DWORD CSimpleEngine::GetColorLeftFront()
{
    return colorLeftFront;
}

//! Gets the color of the back left pyramid vertex
DWORD CSimpleEngine::GetColorBackLeft()
{
    return colorBackLeft;
}

//! Gets the color of the back right pyramid vertex
DWORD CSimpleEngine::GetColorBackRight()
{
    return colorBackRight;
}

//! Gets the color of the back buffer
DWORD CSimpleEngine::GetColorBackBuffer()
{
    return colorBackBuffer;
}
```

Basically, all that `SimpleEngine` does is create an unmanaged Direct3D context, render a scene with a multicolored pyramid, and release the context. In addition to the context and device-specific functions, there are a bunch of accessors that allow you to set and get the color values for various vertices of the pyramid, as well as getting and setting the background color of the device. Later on, we will wrap these accessors into properties within a managed wrapper so we can witness real-time changes to the rendered scene.

Creating a Managed Wrapper for SimpleEngine

A device context in unmanaged Direct3D is provided with a Win32 window handle with which to determine the rendering region for the device. With unmanaged code, you create a new window and get a window handle back, but with Windows Forms, you are working with managed objects. Obviously, you cannot pass a `Form` reference to unmanaged Direct3D, so how can you use Direct3D in a managed application without using managed Direct3D? The .NET framework may use a `Form` class to represent a window, but the underlying architecture still uses the Win32 to create the windows, so a handle for the window must be created somewhere. This handle is a property on the `Form` class called `Handle`, and we can pass that to unmanaged Direct3D when creating a device. With that in mind, we can look into creating a managed wrapper and adapter around our existing unmanaged Direct3D code base (`SimpleEngine`) using C++/CLI.

The wrapper built in this chapter is very simple, and basically consists of a managed class that contains an unmanaged pointer to an instance of `SimpleEngine`. Wrapper methods are provided that invoke calls on the unmanaged class instance. The following code shows the header file for the managed wrapper.

```
#pragma once
using namespace System::Drawing;
using namespace System::ComponentModel;

namespace ManagedSimpleEngine
{
    //! Managed wrapper around the unmanaged SimpleEngine class
    public ref class SimpleEngine
    {
    public:
        //! Constructor
        SimpleEngine();

        //! Destructor
        ~SimpleEngine();

        //! Creates a Direct3D device and context using the
        //! specified window handle and dimensions
        BOOL CreateContext(System::IntPtr window, int width, int height);

        //! Rebuilds the projection matrix of the device when the
        //! context resizes
```

```
        void ResizeContext(int width, int height);

        //! Renders the pyramid scene using the Direct3D render device
        void RenderContext();

        //! Used to release the device and Direct3D context resources
        void ReleaseContext();
```

Rather than wrapping a bunch of get and set methods for the public properties of `SimpleEngine`, we will wrap these methods into properties that consuming languages like C# can access with a clean and familiar syntax. In addition to the properties, the `Category` and `Description` attributes are also specified so that if this class is bound to a `PropertyGrid`, we get useful and descriptive content.

```
    public:
        [Category("Pyramid Colors")]
        [Description("Modifies the top corner vertex color.")]
        property Color ColorTopCorner
        {
            Color get();
            void set(Color color);
        }

        [Category("Pyramid Colors")]
        [Description("Modifies the right front vertex color.")]
        property Color ColorRightFront
        {
            Color get();
            void set(Color color);
        }

        [Category("Pyramid Colors")]
        [Description("Modifies the left front vertex color.")]
        property Color ColorLeftFront
        {
            Color get();
            void set(Color color);
        }

        [Category("Pyramid Colors")]
        [Description("Modifies the back left vertex color.")]
        property Color ColorBackLeft
        {
```

```
            Color get();
            void set(Color color);
        }

        [Category("Pyramid Colors")]
        [Description("Modifies the back right vertex color.")]
        property Color ColorBackRight
        {
            Color get();
            void set(Color color);
        }

        [Category("Context Settings")]
        [Description("Modifies the back buffer color.")]
        property Color ColorBackBuffer
        {
            Color get();
            void set(Color color);
        }

    private:
        //! Points to an instance of the unmanaged
        //! SimpleEngine class
        CSimpleEngine* engine;
    };
}
```

It is important to keep in mind that even though the unmanaged class is instantiated from within a managed class, the unmanaged memory remains unmanaged and must be explicitly released. The destructor can be used to release unmanaged memory, similar to the way memory is released in unmanaged C++.

The following source code describes the implementation details behind the managed wrapper.

```
#include "stdafx.h"
#include "ManagedSimpleEngine.h"

namespace ManagedSimpleEngine
{
    //! Constructor
    SimpleEngine::SimpleEngine()
    {
```

```
    engine = new CSimpleEngine();
}

//! Destructor
SimpleEngine::~SimpleEngine()
{
    if (engine != NULL)
        delete engine;
}

//! Creates a Direct3D device and context using the
//! specified window handle and dimensions
BOOL SimpleEngine::CreateContext(System::IntPtr window, int width, int height)
{
    if (SUCCEEDED(engine->CreateContext((HWND)window.ToPointer(),
                                        width,
                                        height)))
        return TRUE;

    return FALSE;
}

//! Rebuilds the projection matrix of the device when the context resizes
void SimpleEngine::ResizeContext(int width, int height)
{
    engine->ResizeContext(width, height);
}

//! Renders the pyramid scene using the Direct3D render device
void SimpleEngine::RenderContext()
{
    engine->RenderContext();
}

//! Used to release the device and Direct3D context resources
void SimpleEngine::ReleaseContext()
{
    engine->ReleaseContext();
}

//! Gets the color of the top corner pyramid vertex
```

```
Color SimpleEngine::ColorTopCorner::get()
{
    return Color::FromArgb(engine->GetColorTopCorner());
}

//! Sets the color of the top corner pyramid vertex
void SimpleEngine::ColorTopCorner::set(Color color)
{
    engine->SetColorTopCorner(color.ToArgb());
}

//! Gets the color of the right front pyramid vertex
Color SimpleEngine::ColorRightFront::get()
{
    return Color::FromArgb(engine->GetColorRightFront());
}

//! Sets the color of the right front pyramid vertex
void SimpleEngine::ColorRightFront::set(Color color)
{
    engine->SetColorRightFront(color.ToArgb());
}

//! Gets the color of the left front pyramid vertex
Color SimpleEngine::ColorLeftFront::get()
{
    return Color::FromArgb(engine->GetColorLeftFront());
}

//! Sets the color of the left front pyramid vertex
void SimpleEngine::ColorLeftFront::set(Color color)
{
    engine->SetColorLeftFront(color.ToArgb());
}

//! Gets the color of the back left pyramid vertex
Color SimpleEngine::ColorBackLeft::get()
{
    return Color::FromArgb(engine->GetColorBackLeft());
}

//! Sets the color of the back left pyramid vertex
```

```
    void SimpleEngine::ColorBackLeft::set(Color color)
    {
        engine->SetColorBackLeft(color.ToArgb());
    }

    //! Gets the color of the back right pyramid vertex
    Color SimpleEngine::ColorBackRight::get()
    {
        return Color::FromArgb(engine->GetColorBackRight());
    }

    //! Sets the color of the back right pyramid vertex
    void SimpleEngine::ColorBackRight::set(Color color)
    {
        engine->SetColorBackRight(color.ToArgb());
    }

    //! Gets the color of the back buffer
    Color SimpleEngine::ColorBackBuffer::get()
    {
        return Color::FromArgb(engine->GetColorBackBuffer());
    }

    //! Sets the color of the back buffer
    void SimpleEngine::ColorBackBuffer::set(Color color)
    {
        engine->SetColorBackBuffer(color.ToArgb());
    }
}
```

Consuming the Managed Wrapper

The managed wrapper implementation is finished and could be used "as-is," but doing so would result in messy code that probably will be hard to maintain and reuse. We can take our wrapper one step further and wrap the managed `SimpleEngine` wrapper into a `UserControl` that can be dragged onto a form. We are going to avoid extra design-time functionality, because this would lead to problems with device creation during form design. Instead, a `Create()` and `Release()` method will be used to manage device initialization and release. In this way, the regular designer support for a `UserControl` still exists, but we do not have any device-related problems to work through. During device creation, an event handler is attached to

the idle event of the application to support "real-time" rendering, and I say that loosely. Some managed Direct3D implementations use a timer or loop that calls `Application.DoEvents()`, but these solutions are problematic for a variety of reasons. Tying rendering into the idle event of the application seems to be the best approach for now. Aside from the generated designer code, the following source code describes the `SimpleEngine UserControl`.

```
/// <summary>Class that encapsulates the managed SimpleEngine
/// wrapper into a reusable control</summary>
public partial class SimpleEngineControl : UserControl
{
    /// <summary>Reference to the SimpleEngine managed wrapper</summary>
    private ManagedSimpleEngine.SimpleEngine simpleEngine = null;

    /// <summary>Gets or sets the reference to an instance of the
    /// PropertyGrid adapter class</summary>
    public ManagedSimpleEngine.SimpleEngine EngineInterface
    {
        get { return simpleEngine; }
    }

    /// <summary>Constructor</summary>
    public SimpleEngineControl()
    {
        InitializeComponent();
    }

    /// <summary>Destructor \ Finalizer</summary>
    ~SimpleEngineControl()
    {
        Release();
    }

    /// <summary>Creates an engine context and attaches
    /// the necessary events</summary>
    public void Create()
    {
        Release();
        simpleEngine = new ManagedSimpleEngine.SimpleEngine();
        simpleEngine.CreateContext(RenderPanel.Handle,
                                   RenderPanel.Height,
                                   RenderPanel.Width);
```

```
        Application.Idle += new EventHandler(Application_Idle);
        RenderPanel.BackColor = Color.Black;
    }

    /// <summary>Releases the current engine context and
    /// resets the control</summary>
    public void Release()
    {
        if (simpleEngine != null)
        {
            Application.Idle -= new EventHandler(Application_Idle);
            simpleEngine.ReleaseContext();
            simpleEngine = null;
            RenderPanel.BackColor = SystemColors.GradientActiveCaption;
        }
    }

    /// <summary>Event fired when the engine control is resized</summary>
    /// <param name="sender">The sender of the event</param>
    /// <param name="e">The event arguments</param>
    private void RenderPanel_Resize(object sender, EventArgs e)
    {
        if (simpleEngine != null)
            simpleEngine.ResizeContext(RenderPanel.Width, RenderPanel.Height);
    }

    /// <summary>Event fired when application is idle</summary>
    /// <param name="sender">The sender of the event</param>
    /// <param name="e">The event arguments</param>
    void Application_Idle(object sender, EventArgs e)
    {
        if (simpleEngine != null)
            simpleEngine.RenderContext();
    }

    /// <summary>Invoked when the control is supposed to paint</summary>
    /// <param name="e">The arguments of the paint event</param>
    protected override void OnPaint(PaintEventArgs e)
    {
        Application_Idle(this, e);
    }
```

```
    /// <summary>Invoked when the control is supposed to paint
    /// the background</summary>
    /// <param name="e">The arguments of the paint event</param>
    protected override void OnPaintBackground(PaintEventArgs e)
    {
        Application_Idle(this, e);
    }
}
```

Using our new control is very easy; drop it on a `Form` and resize the bounds accordingly. Call the `Create()` method on the control instance to initialize the engine context, and call the `Release()` method when you are ready to destroy the engine context. We have exposed some properties of `SimpleEngine` with accompanying metadata descriptions and category names, so we might as well hook a property grid up to the engine context. The following source code shows example usage of the `SimpleEngine` control and a `PropertyGrid`. In the following example, `EngineContext` is an instance of the `SimpleEngine UserControl`, and `EngineProperties` is an instance of `PropertyGrid`.

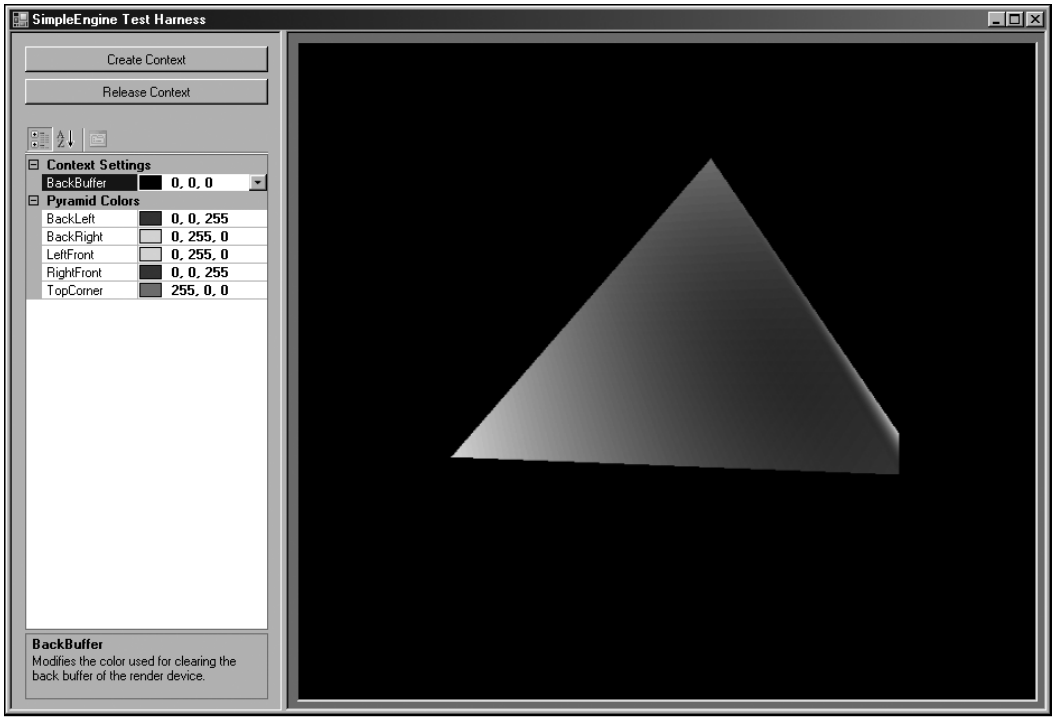
```
public partial class MainForm : Form
{
    /// <summary>Constructor</summary>
    public MainForm()
    {
        InitializeComponent();
    }

    /// <summary>Event fired when the create context
    /// button is clicked</summary>
    /// <param name="sender">The sender of the event</param>
    /// <param name="e">The event arguments</param>
    private void CreateContextButton_Click(object sender, EventArgs e)
    {
        EngineContext.Create();
        EngineProperties.SelectedObject = EngineContext.EngineInterface;
        EngineProperties.Refresh();
    }

    /// <summary>Event fired when the release context
    /// button is clicked</summary>
    /// <param name="sender">The sender of the event</param>
    /// <param name="e">The event arguments</param>
    private void ReleaseContextButton_Click(object sender, EventArgs e)
    {
```

```
            EngineProperties.SelectedObject = null;
            EngineContext.Release();
        }
}
```

Bonus Figure 2.2 shows the SimpleEngine control in action. Notice the `PropertyGrid` that can change engine properties at runtime.



Bonus Figure 2.2 Screenshot of the accompanying example and SimpleEngine control.

Conclusion

This chapter briefly introduced an overview of the C++/CLI language and compiler options, and discussed a few techniques to mix managed and unmanaged code. This chapter does not come close to covering the entire language, and it would be unrealistic to attempt such a goal. The language specification alone is around 300 pages, not factoring in examples and context discussion. Instead, a brief overview was presented that covered a few interesting aspects of the language, later showing a managed wrapper around an unmanaged Direct3D "engine."

Hopefully, you will see the value in C++/CLI if you have to port or migrate your existing legacy code over to the managed platform. Managed wrappers perform and integrate much better than exposed COM interfaces, and provide less legacy support and maintenance. Another great feature of managed wrappers is the ability to refactor some of your code. You can expose a different interface definition than your legacy code exposes, and just wrap the logic accordingly.

C++/CLI is not for everyone, and if possible, you are much better off having no unmanaged code in your application. C++/CLI isn't always the "silver bullet" for many of your interoperability concerns.